



LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

Math Parsing in the Virtual Beam Line Code

B. J. Kraines

August 8, 2014

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Math Parsing in the Virtual Beam Line Code

Benjamin Kraines

August 5, 2014

Abstract

Using a Java Math Parsing Library, several important usability features were added to the Virtual Beam Line (VBL) Code. Equations can represent numerical physical properties of laser beams and other optical components within the code. New sets of global constants allow users to dynamically define parameters and open new possibilities for future laser physics calculations in VBL.

Contents

1	Introduction	2
2	Parsing Math in Code	2
2.1	Human Readable vs. Computer Readable	2
2.2	Dijkstra's Shunting Yard Algorithm	3
3	Math Parsing in The Virtual Beam Line code	5
3.1	Exp4J Library	5
3.2	Shell Class for Equation Parsing	6
4	Results & Use Cases	6
4.1	Beam Profile at Injection	6
4.2	Temporal Profiles	8
4.3	Gain Profiles	9
4.4	Phase Change Application	10
4.5	Constants	10
5	Conclusion	12

1 Introduction

The Virtual Beam Line (VBL) Code is a powerful tool in modeling laser propagation, especially in the context of the National Ignition Facility (NIF). The code provides a quasi-scripting language which enables the user to precisely control all aspects of the simulation, in which the beam travels through space and amongst any number of optical components. This Java code required a number of auxiliary file inputs, which were at times unwieldy and required pre-processing to occur before any calculations could be run within VBL. Adding a new interface capability to replace these file inputs would save time and increase flexibility of input for the code as a whole. In many cases, an equation can be used to represent the data held discretely in auxiliary files. Thus, adding math parsing capability in the code would allow equations to replace the file inputs. Using some well established techniques in Computer Science and readily available libraries to support them, this parsing option was implemented in the VBL code.

2 Parsing Math in Code

A grand challenge in the Computer Science of yesteryear was to apply computational techniques to equations we typically would write and evaluate as humans. Issues of operator precedence and symbolic interpretation were historically barriers to this process. Techniques have been developed to handle these problems with reliability, and are applied in the Equation Parser in VBL.

2.1 Human Readable vs. Computer Readable

There is a stark contrast between the way that a computer might read an expression efficiently, and the way humans do. People generally write equations in infix notation. That is, the operators are interspersed between the values and the equation is read from left to right, but not necessarily evaluated left to right in order. This is a problem for computers, because data structures use a specific order inherently. As such, a new format is required to make an expression computer readable. There are two main expression formats used in today's computer science. The first is prefix notation. It takes the operators and places them to left of the operators to which it applies. This simple example shows an infix expression on the left, translated to a prefix notation on the right.

$$6 + 7 \rightarrow + 6 7$$

The second prominent format for computer readable is just the opposite, postfix notation. This involves placing the operators after the operands to which they apply, as seen in the example below.

$$6 + 7 \rightarrow 6 7 +$$

Once the expression is placed in some computer readable format, a concrete set of rules applies to evaluate them. For example, in the postfix notation the algorithm can simply traverse the expression until it finds an operator, then apply that operator to the previous two operands that it finds. This process is easily scalable to more complex expressions, as seen below in this infix to postfix conversion.

$$5(6 + 7) - 8 \rightarrow 67 + 5 * 8 -$$

In the above example, the algorithm would traverse the postfix representation left to right. It would start by stepping to the first operator, then evaluating the two operands to the left: 6 and 7. Then the algorithm steps through to the multiplication. It looks for two multiplying operands to its left: the result of $6+7$ and the 5. This process can simply iterate until the whole expression is evaluated.

2.2 Dijkstra's Shunting Yard Algorithm

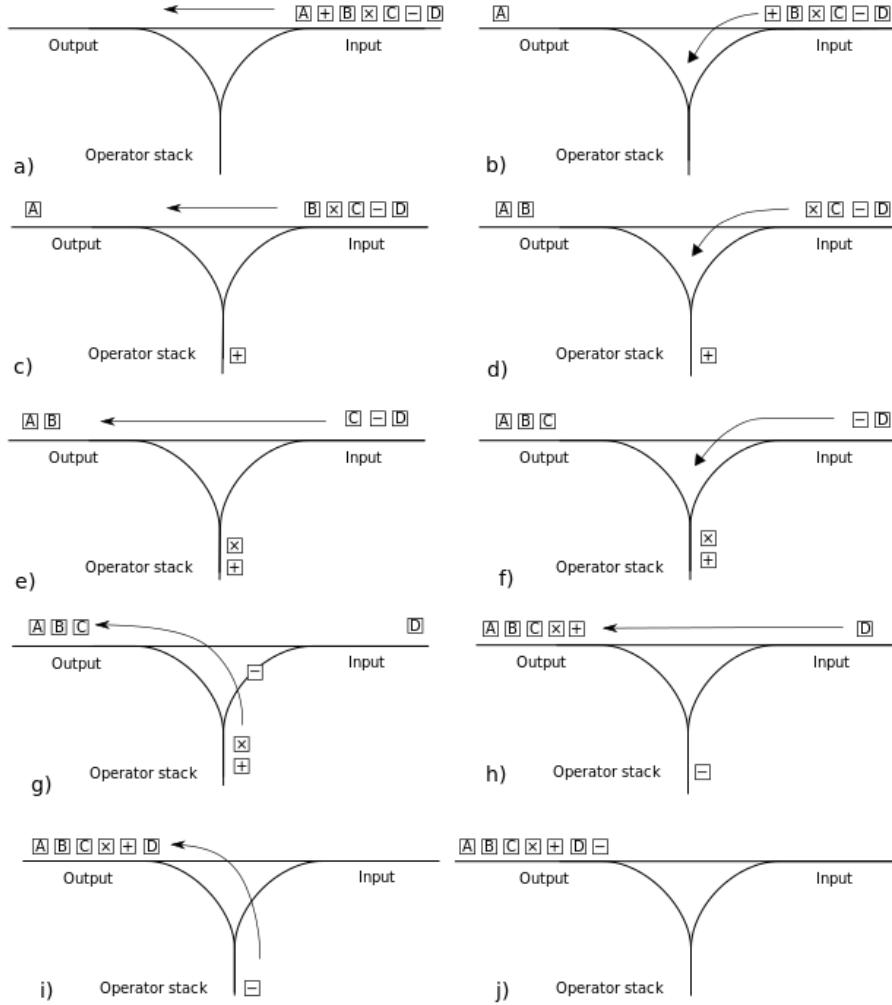
The British term for rail yard is "Shunting Yard". Dijkstra's Shunting yard algorithm solves expressions by rearranging operators and operands much like engineers might rearrange train cars in a rail yard.

Figure 1: *A rail yard.*



Dijkstra envisions a switching track connected to some main line, where train cars can depart from the sequential main line, be rearranged, and then be passed back onto the main line. In this analogy, train cars would be operators and operands in an infix expression on the main line. A stack data structure can represent the switching track, as it is a Last-In-First-Out (LIFO) structure. Operators are always pushed onto the stack, and operands always follow the "main line". Then Dijkstra uses some logic blocks to sort out when the operators need to get pushed in with the operands to generate postfix. The basic idea can be shown in the following diagram.

Figure 2: *Shunting Yard Algorithm Visualization*



The diagram above shows the basic data flow as the algorithm processes a simple algebraic statement from infix notation. As the program iterates over the expression from left to right it utilizes parenthesis and a preset operator hierarchy that determines when the operator gets pushed off the stack. In the example above, we see that the multiplication operator has the highest precedence. Hence it gets popped along with the addition after step d, when the subtraction operator appears at the switch. The addition has an equal precedence, but earlier time of arrival than the subtraction operator, so it gets pushed to the output expression as well. This process continues until the whole expression is transformed into infix notation. Once the expression is transformed fully into postfix, then the algorithm traverses it, evaluating postfix, as described in section 2.1.

3 Math Parsing in The Virtual Beam Line code

The Virtual Beam Line code's primary data structure is physical field values on a spatial grid. As laser beams propagate in the system, they occupy a space in the x,y, and z axis. Parameters of the beam's state can be characterized by the x/y position of the point under consideration. For example, a phase distribution might characterize the phase at each x/y pair in the beam's grid for a given beam state. In addition, the components with which the beam interacts are similarly defined by spatial grids. Slab amplifiers for the laser have a unique gain distribution that amplifies the beam differently based upon the location that part of the beam might be incident upon the slab spatially. Traditionally, these distributions had to be manually inputted using discrete auxiliary files, even though these distributions can be represented by continuous functions. Adding a capability to VBL for transforming these continuous math functions into the discrete distributions already used provides a flexible new way for users to experiment and provide input in a new way.

3.1 Exp4J Library

One open source library for Java which supports expression parsing is the Exp4J (Expressions for Java) Library. It features the previously discussed Shunting Yard Algorithm and provides a very helpful Application Programming Interface (API) to access this parsing capability. The library features modifiable operator precedence options, custom functions, flexible variables, and easy integration. It stands out among other Java math parsing libraries in these respects. Some other libraries were tested as potential options for this role, including ExpressionOasis, and Java Expression Parser. Exp4J ended up being the most powerful math parsing tool available that would fit well within the VBL code structure.

3.2 Shell Class for Equation Parsing

In order to make the Exp4J library useful, VBL needed a shell class that handles all math expression evaluation. A class called EquationParse was created to fill that need. At any time, VBL can instantiate an EquationParse object and use it to process a given string containing an infix expression. The shell class holds a series of member variables which govern the specifics of evaluating functions. Member data includes the infix string input, a variable dictionary, a calculation object from Exp4J, and a VBL Path object. The EquationParse shell class also holds a series of utility functions that assist the VBL objects that use the Equation Parser. These include functions to create distributions over specific x/y grids, utilities to write aux files out for the user, and other distribution related numerical functions.

4 Results & Use Cases

Math parsing was successfully implemented into VBL with a number of important new use cases for laser physics calculations. Each different application uses the same EquationParse shell class to perform its unique action set in the context of equations.

4.1 Beam Profile at Injection

One of the first things VBL does is inject a beam into the beamline. This involves specifying the shape, energy, size, and other parameters of the initial beam. Previously, the only ways to do this was using a template square or Gaussian beam profile, or using an auxiliary file. Equations supplement these input options by allowing the spatial definition of fluence at the input. From this distribution, the shape can be scaled to a new energy, or the inherent energy of the distribution can infer the total energy of the beam. An example of creating a new equation defined injection fluence profile might look like this:

Figure 3: *Beam injection statement using equations.*

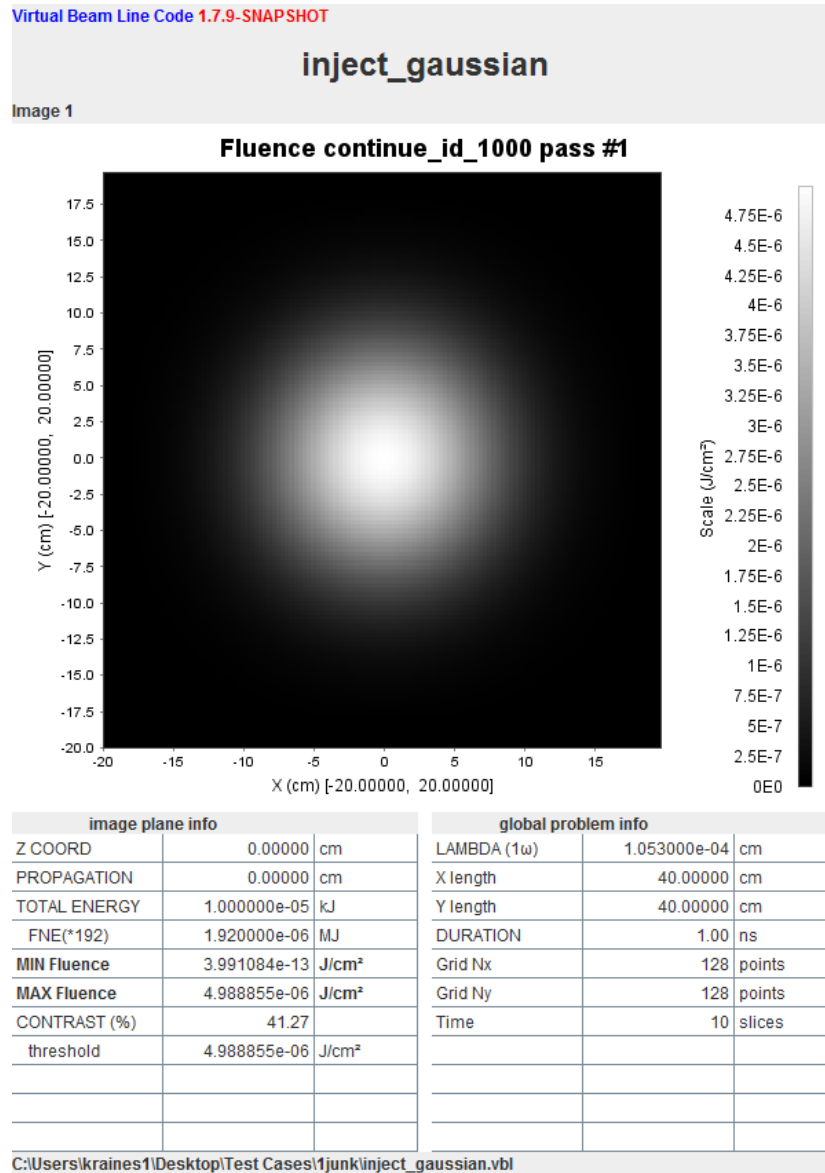
```
inject eqn=[exp(-(x/8)^2 -(y/8)^2)]
          tpulse=1.0
          energy = 100e-04
```

This statement would inject a Gaussian beam according to the equation:

$$e^{-\frac{x^2}{8} - \frac{y^2}{8}}$$

VBL would then scale this profile such that the total energy is equal to 100e-04 kJ, as specified by the energy keyword. The pulse would last one nanosecond, per the tpulse parameter. The beam would look something like Figure 4 at VBL's fluence output, placed just after injection.

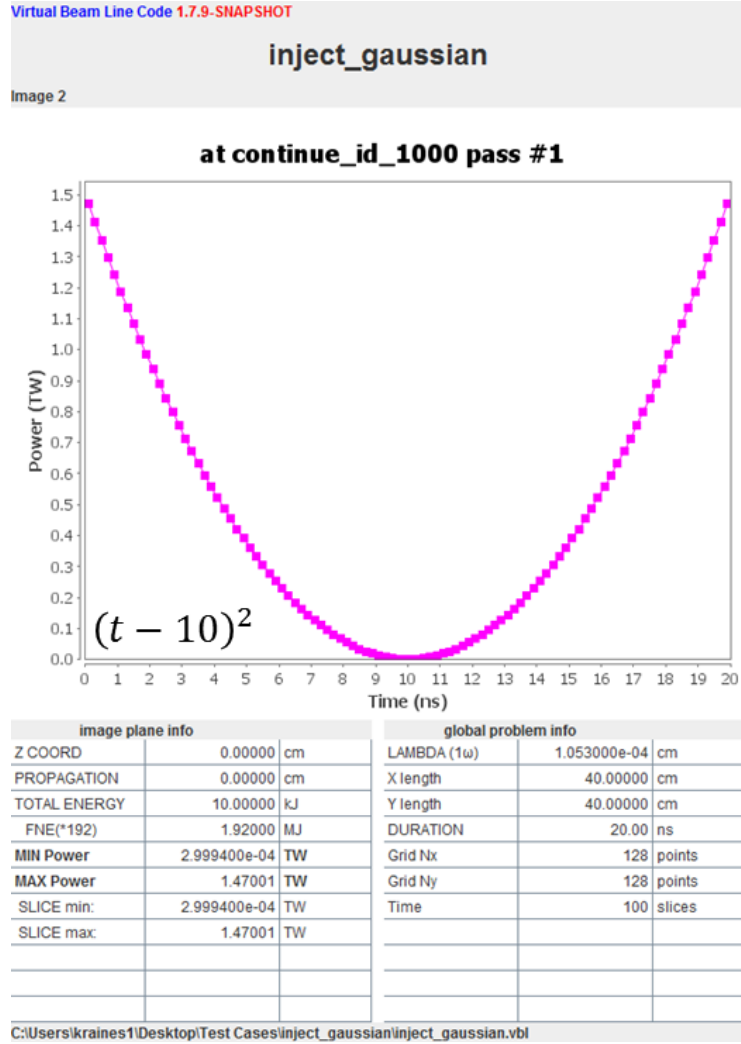
Figure 4: VBL output after equation injection.



4.2 Temporal Profiles

In addition to specifying the fluence profiles of beams, the power of a pulse over time can be shaped using an equation. In the NIF, the pulse shape is an extremely important characteristic of the beam. Thus, being able to precisely control and experiment with this characteristic in simulation might be helpful.

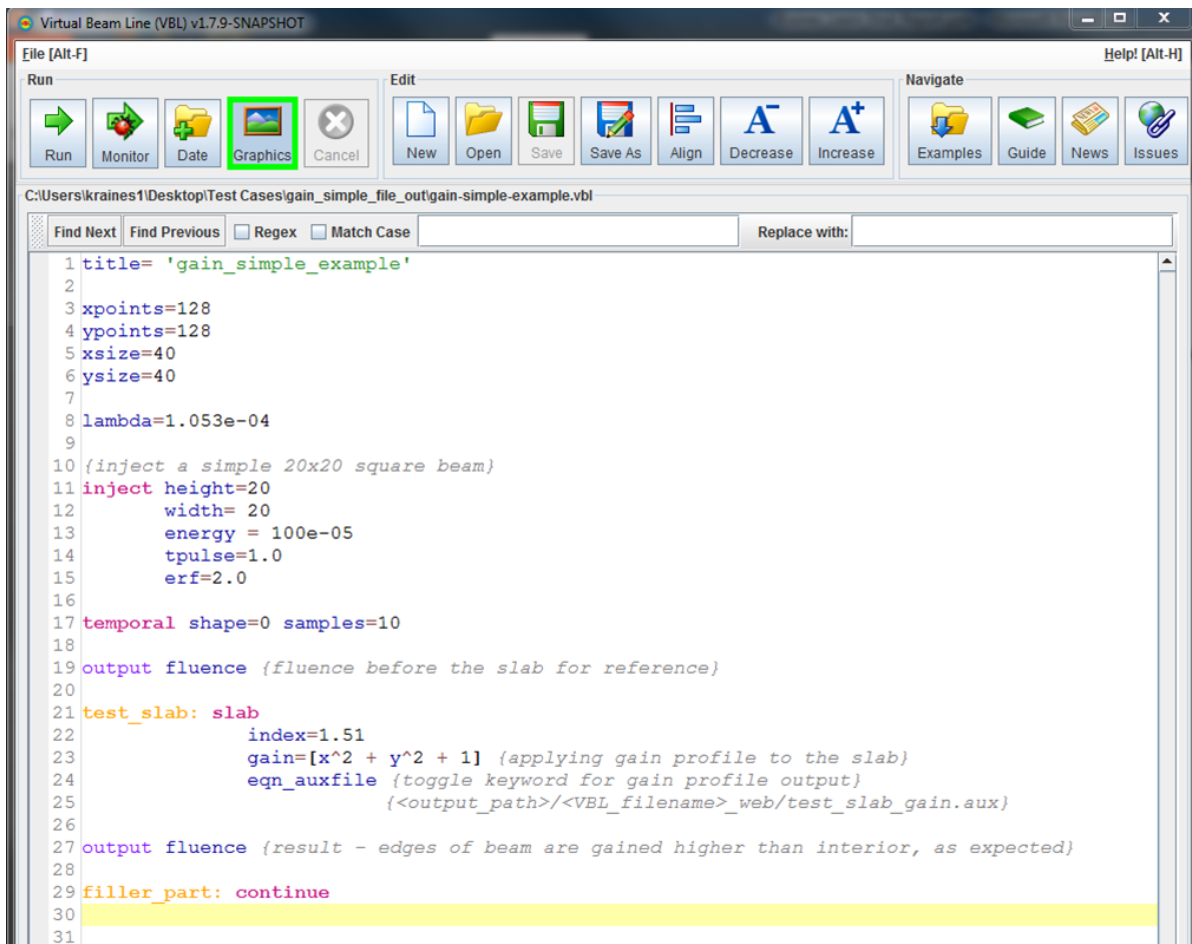
Figure 5: A temporal power trace from equations in VBL.



4.3 Gain Profiles

When a beam passes through a slab amplifier, the laser gains energy often times in a non-uniform way. Some parts of the beam might absorb more energy than others. For this reason, when simulating the amplifiers, specified gain profiles must be interpolated to represent the distribution of gain spatially on a beam incident to the slab. A simple example of how to specify a gain profile equation in VBL is shown in Figure 5.

Figure 6: A deck utilizing equation gain profiles in VBL.

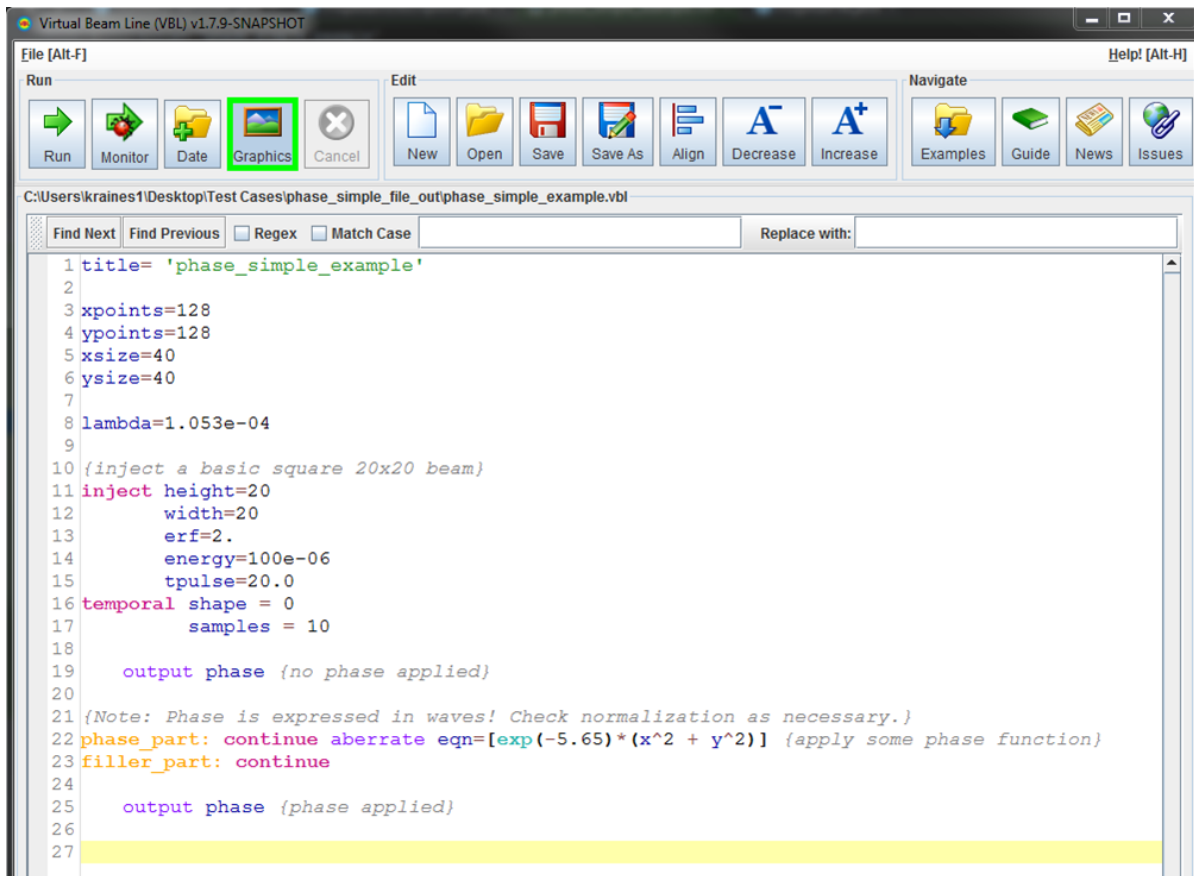


```
1 title= 'gain_simple_example'
2
3 xpoints=128
4 ypoints=128
5 xsize=40
6 ysize=40
7
8 lambda=1.053e-04
9
10 {inject a simple 20x20 square beam}
11 inject height=20
12     width= 20
13     energy = 100e-05
14     tpulse=1.0
15     erf=2.0
16
17 temporal shape=0 samples=10
18
19 output fluence {fluence before the slab for reference}
20
21 test_slab: slab
22     index=1.51
23     gain=[x^2 + y^2 + 1] {applying gain profile to the slab}
24     eqn_auxfile {toggle keyword for gain profile output}
25                 {<output_path>/<VBL_filename>_web/test_slab_gain.aux}
26
27 output fluence {result - edges of beam are gained higher than interior, as expected}
28
29 filler_part: continue
30
31
```

4.4 Phase Change Application

As a laser beam moves through optics, it often accrues phase changes that can affect the way that it propagates. These are called aberrations. In a beam defined spatially, a phase applied to that beam is also defined spatially. Thus in the application of an aberration, an equation can be entered in terms of x and y to change the beam's phase. An example of how to specify a phase equation in VBL is seen in Figure 6.

Figure 7: A deck utilizing equation aberration profiles in VBL.

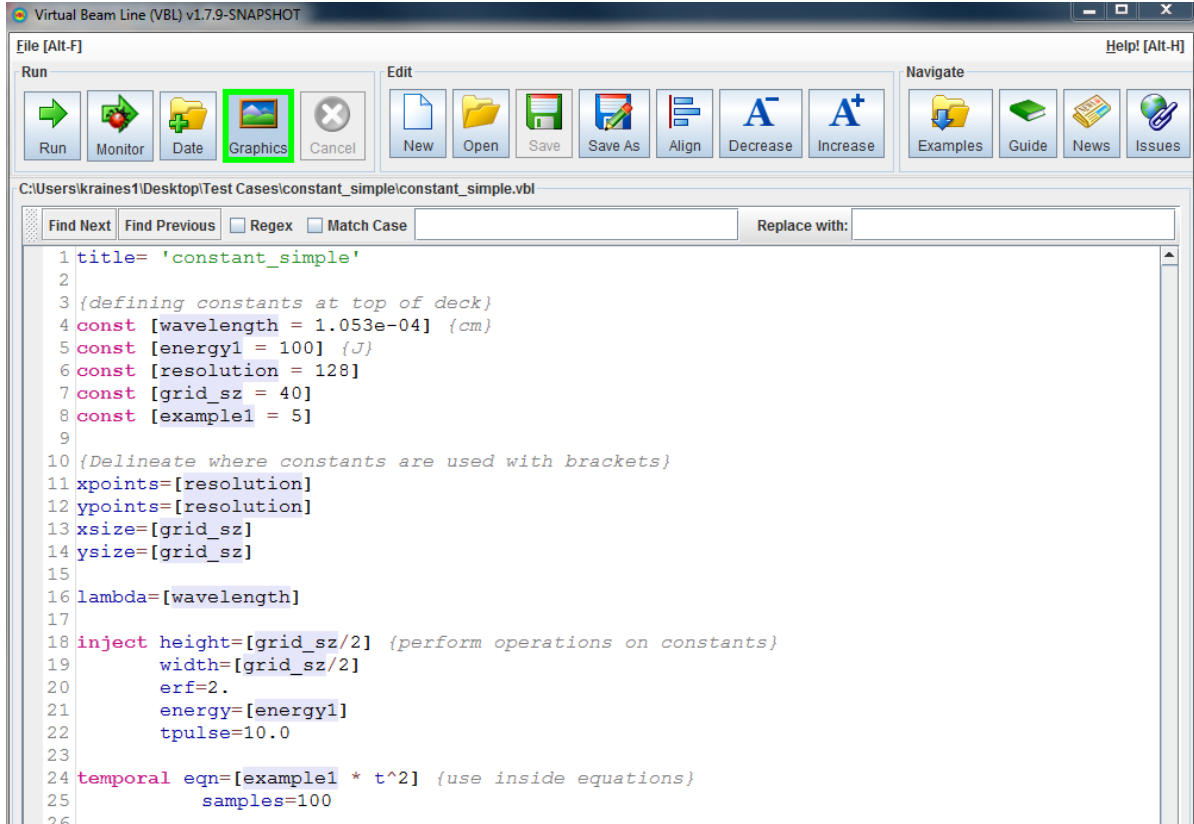


4.5 Constants

When creating large VBL decks using many parameters, it becomes cumbersome to repeatedly enter the same constants in different components throughout the

deck. As a user, it is easy to lose track of the significance for the literal numerical values. By adding constants, VBL provides a structure to maintain context and assist in deck stewardship. Previously in VBL, macros were present to perform text replacement and generally support this function. However since macros only do simple text replace, they weren't dynamic within the code. Global constants can provide this flexibility. Using the EquationParse shell, constants can now be defined in VBL at the beginning of the deck, and then used later with math operations. Anywhere in the code, the user can open a set of brackets and write a constant, manipulated by any math they so choose. An brief example showing global constant usage is seen in Figure 7.

Figure 8: *A deck utilizing global math constants in VBL.*



5 Conclusion

Adding math parsing and global constants to the Virtual Beam Line code increases the flexibility of input and use for simulating laser propagation. Users can now enter equations to represent physical characteristics of laser beams or optical components when creating new VBL decks. These equations are translated to numerical distributions using a parsing library featuring Dijkstra's shunting yard algorithm. Constants can be placed anywhere in a VBL deck and be parsed as is to increase the flexibility and ease of problem setup for users. Overall, the new math parsing abilities of VBL extend the usability and function of the code in a unique way.

References

- [1] R.A.Sacks, E. Feigenbaum, K.P. McCandless, D. Potter.
VBL User Guide v1.7.9
Laser Performance Modeling Group, National Ignition Facility
April 2014
- [2] Frank Asseg
Exp4J 0.4.0 and API Guide Apache License, Version 2.0
<http://www.objecthunter.net/exp4j/>
- [3] Saliz Alba
"Shunting Yard"
Licensed under Creative Commons Attribution-Share Alike 3.0
http://commons.wikimedia.org/wiki/File:Shunting_yard.svg
- [4] Arne Huckelheim
"Frankfurt Central Station Tracks"
Licensed under Creative Commons Attribution-Share Alike 3.0
<http://upload.wikimedia.org/wikipedia/commons/2/26/SunsetTracksCrop.JPG>

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.